Subject: Minor things? left over from 00-144r1, and a few more
From: Van Snyder
References: 00-103 00-136r1 00-144r1

1 Introduction

The issues mentioned in 00-103 that didn't get addressed were collected into 00-144r1. Part of 00-144r1 was studied at meeting 152, and part was not. Of the part that was studied, several edits were passed. I collected these, together with the part that was not studied, into 00-144r1.

Of the part that was not studied, I thought that several were probably typos, and part probably had technical content. I divided these into separate sections (2 and 3) in 00-144r1.

The editor (taking a suggestion I wrote into 00-144r1) has studied the ones I thought were typos, accepted some, and rejected others (some on stylistic grounds, some because they had technical content). Four of those are included here, with more explanation, for consideration at meeting 153.

There were eight edits that were identified in 00-144r1 by "may have technical content." Of those, six are repeated here for consideration at meeting 153.

I've also added a few.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

2.1 From part 2 of 00-144r1

183:8-12 [Editor: The part from "A derived-type object" to the end of the note applies to normative text at [183:36]. Split the note into two parts, and move the identified part to [183:36+], leaving it as a nonnormative note.]

253:46 [Editor: We are careful in other places, e.g. (7.5.2) and (5.1.2.11), not to suggest that all intrinsic functions can be used as actual arguments. E.g., even though AMAX1 is a specific name of an intrinsic function, it cannot be used as an actual argument. So as not to contradict other sections, make the wording here the same as is used in (7.5.2) and (5.1.2.11), i.e. replace "function" by "procedure listed in 13.15 and not marked with a bullet (•)" after "function" (but don't do this if 00-187 passes).]

261:4 [The invoking program unit may not be a subprogram. Editor: Replace "subprogram" by "scoping unit." The term "invoking scoping unit" is used frequently in (5.1.2.3), and possibly elsewhere.]

2.2 From part 3 of 00-144r1

[Editor: Add "and type parameters" after "array bounds."]	105:44
[Editor: Replace "is determined by the constructor name" by "and type parameters are as described in 4.5.6".]	115:11-12
[Editor: After "are" add "those of the specific function referenced,"; after "arguments" add ", as specified in 14.1.2.3".]	115:16-17
[Editor: Before "from" insert ", accessed by use association". After "IEEE_ARITHMETIC" add "(15)".]	119:14-15
[Editor: Remove "or" at [259:48]. Insert ", or a reference to the NULL() intrinsic" after "pointer" at [260:1].]	259:48-1
[Editor: Replace "the vendor ... support" by "the supported subset of features is processor dependent." These kinds of things are usually (always?) described by reference to the processor, not the vendor.]	361:35-36

2.3 New ones

[The example is incorrect. The second argument does not have the same characteristics as the corresponding argument in the procedure `POINT_3D_LENGTH` overrides. Editor: Replace by the following:]
`CLASS (POINT_3D), INTENT(IN) :: A`
`CLASS (POINT_2D), INTENT(IN) :: B`
	54:38

[Editor: Add PARAMETER to the list. Also see edit for [81:29] below.]	64:23
[I couldn't find a constraint against ALLOCATABLE and EXTERNAL being specified for the same entity. Replace the one about ALLOCATABLE and POINTER by:] Constraint: If an entity has the ALLOCATABLE attribute it shall not have the POINTER or EXTERNAL attribute.	64:37-38
[Editor: Delete "PARAMETER," and "DIMENSION,". The PARAMETER attribute already can't be applied to dummy arguments, which is the only place VALUE can be used. The DIMENSION attribute is prohibited by the constraint at [65:21-22]. But don't bother with this if 00-170 passes.]	65:15,16
[Editor: Capitalize "external" twice to be consistent with usage in the other three paragraphs in this section.]	76:31, 33
[Editor: Delete – Duplicates the constraint at [71:11-12]. See [79:21-23].]	79:27
[Editor: Delete – Duplicates the constraint at [64:31-32]. See [79:21-23].]	80:26-28
[Editor: Delete – Duplicates the constraint at [64:18-20]. See [79:21-23].]	81:17-18
[Editor: Delete – Duplicates the constraint at [64:18-20]. See [79:21-23].]	81:27-28
[Editor: Delete – Duplicates the constraint at [64:23], as modified above. See [79:21-23].]	81:29
[Editor: Delete – Duplicates the constraint at [64:25-26]. See [79:21-23].]	81:38
[Editor: Delete – Duplicates the constraint at [65:32]. See [79:21-23].]	83:1-2

[Editor: Delete – Duplicates the constraint at [64:19-20]. See [79:21-23].]	91:15-16
[This repeats material at [134:40], but the intent here appears to be list all the circumstances in which things get deallocated.] When an intrinsic assignment statement (7.5.1.5) is executed, allocatable components of the *variable* are deallocated before the assignment takes place.	107:27+
[Editor: Insert "is" after the first "and", delete the second "and".]	118:13
[Editor: "When" ⇒ "If".]	189:31
[Editor: Start a new paragraph] The `err`, `eof`, and `eor` arguments correspond, respectively, to the ERR= (9.9.3), END= (9.9.4) and EOR= (9.9.5) specifiers in a data transfer input/output statement.	192:27+
[Editor: "hargument" ⇒ "argument".]	237:22
[Editor: Replace "OPTIONAL ... PUBLIC" by "or OPTIONAL" – Duplicates the constraint at [70:40]. See [79:21-23].]	242:2
[Editor: Insert "(functions only)" at the end – for consistency with the other ones in the list.]	245:27
[Editor: Insert "7.1.3" before "7.1.8.7". 7.1.3 seems to be the reference for defined operations that would be most interesting here.]	249:23
[Editor: Insert "and type-bound procedure bindings" at the end.]	251:37
[The *proc-decl-list* isn't optional. Editor: Replace "at most" by "exactly".]	252:42
[Editor: Simplify by inserting "those" after "are" and deleting "and are ... interface".]	253:9
[Editor: After "argument" insert "other than the passed-object dummy argument".]	255:10
[Editor: After "argument" insert "that does not have INTENT(IN)".]	257:11
[Editor: After "pointers" insert "that do not become undefined and are". Otherwise, the statement implies that local pointers without the SAVE attribute retain their values if they happen to be associated with a dummy argument that has the characteristics discussed at this point.]	257:41
[Editor: After "dummy argument" insert "type". A type can't be an extension of a dummy argument.]	258:30
[There is a constraint at [255:28] that is nearly identical to the sentence "The label ... reference." Here, it says "executable construct" while in the constraint at [255:28] it says "branch target." I don't know if there's a real inconsistency, but the difference in wording is confusing. The simplest solution is to remove the sentence here. Editor: Insert "(12.4)" after "specifier" and delete the sentence "The ... reference."]	260:19-21
[Editor: Delete "that" (finish – hopefully – work begun in 00-136r1).]	352:18

Subject: Syntax and Semantics of VALUE attribute, issue 214, part of issue 90
From: Van Snyder
References: 00-171

1 Introduction

I propose that the syntax for what is now called the VALUE attribute ought to be INTENT(VALUE), and that the semantics should be that the subprogram can change the dummy argument, but the associated actual argument is not thereby changed. This is the way it works in C if the asterisk is omitted from a formal argument.

The reasons for this proposal are

- The semantics of the VALUE attribute are more different than necessary from the semantics of omitting the asterisk from a dummy argument in C,
- The semantics of the VALUE attribute are less useful than they would be if they were more similar to the semantics of omitting the asterisk from a dummy argument in C, and
- The standard would be simpler if the syntax were INTENT(VALUE): Almost everywhere the VALUE attribute is mentioned, the INTENT attribute is mentioned in the same sentence. If INTENT(VALUE) were used, the prohibition against duplicate specification would remove the need for any discussion of the relation between VALUE and INTENT.

Malcolm says the semantic change proposed here doesn't work because it runs afoul of the association rules in 12.4, 14.6.1.1, and item 12 in 14.7.5. Malcolm's "easy fix" is to use INTENT(VALUE) instead of VALUE. Then, all of the constraints that say "... VALUE, INTENT..." can just say "... INTENT...", and everywhere that says "INTENT(IN) and/or VALUE" would say "INTENT(IN) and/or INTENT(VALUE)".

I've put my original proposal for the semantics of INTENT(VALUE) as a separate section, so it's written down in case somebody really likes it and figures out how to make it work, and put in Malcolm's idea instead. Everything else is the same either way. The only justification that remains is the third point above.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Delete "5.2.12 VALUE statement".]	iii
[Editor: Delete the syntax rule for the VALUE attribute.]	64:7
[Editor: Delete ", VALUE," because INTENT covers it.]	64:29

[Editor: Delete. Re-stated by the edit at [71:19+] below.] 65:15-20

The constraint at [65:15-17] includes a prohibition against PARAMETER, which isn't needed because the PARAMETER attribute is prohibited for dummy arguments at [63:27], and a prohibition against DIMENSION, which is prohibited at [65:21]. So even if the change proposed here is not accepted, the prohibitions against PARAMETER and DIMENSION should be removed from the constraint at [65:15-17]. *Note to J3*

[Editor: Delete. There's no reason for it.] 65:21-22

[Editor: Replace "VALUE" by "INTENT(VALUE)"' twice.] 65:28, 31

or VALUE 71:10+

Constraint: If the INTENT(VALUE) attribute is specified for a dummy argument of a subprogram or interface body that has a *language-binding-spec* 71:19+

(1) The dummy argument shall be a scalar, and

(2) If the dummy argument is of character type, the length parameter shall be omitted or shall be specified by an initialization expression with the value one.

Constraint: If the INTENT(VALUE) attribute is specified the ALLOCATABLE, POINTER, or VOLATILE attribute shall not be specified.

There is no need to prohibit EXTERNAL because that's covered by the constraint at [71:11-12]. If the change in 00-171 is accepted, a conspiracy of the revised constraint and the constraint against POINTER serves. *Note to J3*

The INTENT(VALUE) attribute implies the INTENT(IN) attribute. A processor may choose, however, to use different argument passing mechanisms for INTENT(IN) and INTENT(VALUE) dummy arguments. 71:38+

Note 5.11$\frac{1}{2}$

The name of the INTENT(VALUE) attribute is intended to be suggestive. Although the processor is not required to use pass-by-value for an argument with the INTENT(VALUE) attribute, that might be a possible implementation. If the INTENT(VALUE) attribute is specified for a dummy argument of a procedure or interface body that has a *language-binding-spec*, the processor shall use the same argument passing convention as the companion processor, which is often pass-by-value.

[Editor: Delete section 5.1.2.14, including unresolved issue 214.] 78:15-35

[Editor: Delete.] 82:31-33

[Editor: Replace "VALUE (5.1.2.14)" by INTENT(VALUE) (5.1.2.3)".] 244:16

[Editor: Insert ", whether it has the INTENT(VALUE) attribute" after "pointer".] 244:23

[Editor: Replace "VALUE" by "INTENT(VALUE)".] 245:21

[Editor: Replace "the VALUE attribute" with "INTENT(VALUE)".] 257:36

[Editor: Replace "the VALUE attribute" with "INTENT(VALUE)".] 258:7

[Editor: Replace "`VALUE`" by "`INTENT(VALUE)`".] 390:24

[Editor: Replace "the VALUE attribute" by "INTENT(VALUE)" twice.] 392:33,35

[Editor: Replace "The VALUE attribute" by "INTENT(VALUE)".] 393:8

[Editor: Replace "`VALUE`" by "`INTENT(VALUE)`)" twice.]	393:37,40
[Editor: Replace "the VALUE attribute" by "INTENT(VALUE)".]	393:46
[Editor: Delete "VALUE" from the index twice.	467

3 What I originally had in mind

These have the form of edits, but are no longer part of the proposal.

Constraint: If the INTENT(VALUE) attribute is specified the ALLOCATABLE or POINTER attribute shall not be specified.	71:19+
The INTENT(VALUE) attribute for a nonpointer dummy argument specifies that if the value of the dummy argument is changed or becomes undefined during execution of the procedure, the associated actual argument is not affected. The INTENT(VALUE) attribute for a pointer dummy argument specifies that if the association status of the pointer changes or becomes undefined during execution of the procedure, the pointer association status of the associated actual argument is not affected.	71:38+
The constraint at [65:15-17] originally included a prohibition against VOLATILE. If the semantics are changed as suggested in the edit for [71:38+], there isn't a problem with the VOLATILE attribute: It would apply to the dummy argument, not the associated actual argument. The same reasoning applies to the comments about ASYNCHRONOUS in issue 90 at [65:28-31].	*Note to J3*

Subject: More little problems with procedure pointers
From: Van Snyder

1 Introduction

(1) It ought to be allowed to specify intent for dummy procedure pointers.

(2) Whether a dummy procedure is a pointer ought to be a characteristic.

(3) It ought to be possible to assign a procedure to a procedure pointer with explicit interface from within itself. It's not always possible because the only procedures that have explicit interfaces from within themselves are recursive subroutines and recursive functions that have a result variable different from their names.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.– Items 1 and 2

Constraint: The INTENT attribute shall not be specified for a dummy procedure that is not a dummy procedure pointer. 71:11-12

[Editor: After "explicit," insert "whether it is a pointer,".] 244:23

3 Edits – Item 3

This one is done separately so we can vote on it separately. Maybe there is a subtle reason for the status quo. Malcolm doesn't like this proposal because prohibiting nonrecursive procedures to have explicit interfaces from within themselves makes it difficult to use them as dummy arguments from within themselves, and therefore difficult to cause an erroneous recursion by mistake.

[Editor: Delete "recursive" twice.] 245:4-5

Subject: Issue 263
From: Van Snyder
References: 00-179

1 Introduction

The editor doesn't like the wording of section 2.2.3.4. Here's different wording.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

The purpose of an **interface block** is to describe the interfaces (12.3) to a set of procedures, and the forms of reference by which a procedure may be invoked (12.4). It may be used to specify that a procedure may be invoked: 13:4-7

(1) By using a generic name,

(2) By using a defined operator,

(3) By using a defined assignment, or

(4) For derived-type input/output.

[but don't do the last one if the "radical proposal" in 00-179 passes.]

[Editor: Delete issue 263.] 13:8-12

Subject: Issue 262
From: Van Snyder

1 Introduction

The editor notes that the usage of the term *intrinsic* is contradictory.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: after "**intrinsic**" insert "has two meanings. The first".] 19:34

[Editor: At the end of the paragraph, insert "The second use of the qualifier applies to procedures that are provided by a processor but whose names are not listed in section 13.13, 13.14, 13.15, or modules that are provided by a processor but whose names are not listed in section 13.17, 15 or 16.1. Such procedures and modules are called *nonstandard intrinsic procedures* and *nonstandard intrinsic modules*, respectively." Editor: Should *nonstandard intrinsic procedures* and *nonstandard intrinsic modules* be set in bold face type? If so, are they required to have index and glossary entries?] 19:37

[Editor: Delete issue 262.] 20:1-7

Subject: Issue 236
From: Van Snyder

1 Introduction

The editor finds the placement of the discussion of the kind of type parameter values to be misplaced, and the transition from discussion to syntax to be clumsy.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Move [32:33-34] to [55:25+], and insert the following in its place:] 32:33-34
A type parameter value may be specified within a type specification (5.1).

[Editor: Delete issue 236.] 33:1-16

Subject: Issue 268
From: Van Snyder

1 Introduction

The editor finds the placement of two constraints confusing. The proposed repair is to move two syntax rules, instead.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Delete issue 268.]	43:11-19
[Editor: Move to [42:41+].]	43:23-24
[Editor: Move to [42:43+].]	43:25-26

Subject: Issue 269
From: Van Snyder

1 Introduction

The constraints on passed-object dummy arguments are intentionally slightly different, because they apply in different circumstances. One applies to procedure pointer components, one applies to abstract interfaces, and one applies to a procedure named in a binding.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Copy this sentence to [44:25+].]	44:9
[Editor: Copy the constraint at [44:32-33] to here.]	44:9+
[Editor: set "**passed-object dummy argument**" in bold face type – this is another definition.]	44:24
[Editor: Move this constraint to [45:4-].]	44:32-33
[Editor: Delete issue 269.]	44:34-42
[Editor: Move to [45:3+].]	44:43
[Editor: After "*procedure-name*" insert "or procedure name implied by *binding-name* if *binding* is not specified".]	44:46
[Editor: set "**passed-object dummy argument**" in bold face type – this is another definition.]	45:2

Subject: Issue 7
From: Van Snyder

1 Introduction

The editor is correct that the last sentence of the paragraph at [104:21-24] is redundant to material in 6.3.3.1, and not quite correct anyway.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Delete the sentence beginning "If the object...."]	104:21-24
[Editor: Delete issue 7.]	104:25-29

Subject: More work on SELECT TYPE and ASSOCIATE
From: Van Snyder

1 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Insert "other than a SELECT TYPE (8.1.4.1) or ASSOCIATE (8.1.5.1) statement" after "statement". (We don't want the selector deallocated before the block is executed.)] 107:21

[Now handle the selector.] 107:24+
If a SELECT TYPE (8.1.4.1) or ASSOCIATE (8.1.5.1) statement references a function whose result is allocatable or a structure with a subobject that is allocatable, and the function reference is executed, an allocatable result and any subobjects that are allocated allocatable entities in the result returned by the function are deallocated after execution of the construct.

[Editor: Replace "type" by "declared type, dynamic type, and" (Make it clear that assuming the type means assuming both the declared and dynamic type).] 154:21

Subject: Derived-type input/output and its relation to type-bound procedures
From: Van Snyder
References: 00-186

1 Introduction

There are several problems with derived-type input/output that could be addressed by a minor extension to the type-bound procedure mechanism: It is possible to access a type from a module, but not access its input/output procedures, and it is not possible to inherit, override or defer a derived-type input/output procedure.

This paper depends on paper 00-186. It should be processed, if at all, after that paper passes, or another one that specifies how kind type parameters interact with type-bound procedure invocation passes.

2 Proposed change

Replace the interface-block-based mechanism to specify derived-type input/output procedures by a variation on the type-bound procedure declaration mechanism, e.g.

```
PROCEDURE, READ(FORMATTED) => my_read_routine_formatted
```

serves the same purpose as the interface block at [190:41]. Obviously similar extensions provide for unformatted input and for output. This binding cannot be excluded during use association, is inherited into extension types, can be overridden in them, and an obvious variation allows deferred derived-type input/output procedures to be specified.

The advantage of this approach is that it solves the problems noted above. The disadvantage (or another advantage, depending on your point of view) is that it does not allow a derived-type input/output procedure to be a dummy argument or a procedure pointer.

3 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

44:18+

or PROCEDURE (*proc-interface-name*), ■
■ *dtio-binding-attr* => NULL()
or PROCEDURE, *dtio-binding-attr* => ■
■ *dtio-binding-list*

44:25

[Editor: Add to the constraint:]
If *proc-interface-name* and *dtio-binding-attr* are both specified, the interface shall specify a subroutine having an explicit interface as specified for the same *dtio-binding-attr* in section 9.5.4.4.3.

45:3+

R442b *dtio-binding-attr* **is** READ(FORMATTED)

or READ(UNFORMATTED)
or WRITE(FORMATTED)
or WRITE(UNFORMATTED)

R442c *dtio-binding* **is** *procedure-name*
or NULL(*procedure-name*)
or NULL(*procedure-pointer-name*)

Constraint: The *procedure-name* shall specify an accessible module procedure or external procedure. The *procedure-pointer-name* shall specify an accessible procedure pointer. The *procedure-name* or *procedure-pointer-name* shall have an explicit interface as specified for the same *dtio-binding-attr* in 9.5.4.4.3.

Constraint: If several specific or deferred (4.5.1.5) procedures are specified for a single *dtio-binding-attr*, their interfaces shall differ as specified in 14.1.2.3.

Note 4.19$\frac{1}{2}$

The interfaces are specified nearly completely in section 9.5.4.4.3. The only latitude for differences is that, for a particular type and *dtio-binding-attr*, the derived type dummy arguments can have different kind type parameters.

4.5.1.5.2 Derived-type input/output subroutine 49:30+

A procedure binding with a *dtio-binding-attr* specifies a user-defined derived-type input/output subroutine. Its use and the characteristics it shall have are described in 9.5.4.4.3. The set of user-defined derived-type input/output subroutines that are bound to the type or that are inherited (4.5.3.1) from the parent type and not overriden (4.5.3.2) is a generic interface.

[Editor: Insert a new paragraph, not within note 4.44:] 54:45+
If a procedure binding declared in a type definition has the same *dtio-binding-attr* and the same kind type parameters for the derived-type argument as one inherited from the parent type then the binding declared in the type overrides the one inherited from the parent type. Otherwise it extends the generic interface for the declared type and specified *dtio-binding-attr*.

[Editor: Replace "any procedure ... matches" by "an external or module subroutine, bound to the type by a procedure binding with a *dtio-binding-attr* as specified in section 4.5.1. Its interface shall match".] 189:26

[Editor: Replace "procedure" by "subroutine".] 189:28

[Editor: After "transferred" insert ", as described in 14.1.2.4.2$\frac{3}{4}$".] 189:30

[Editor: Replace "When an interface ... scoping unit" by "If a derived-type input/output procedure is selected as specified in the previous paragraph".] 189:31-32

(1) If the *dtio-binding-attr* is READ (FORMATTED): 190:26

(2) If the *dtio-binding-attr* is READ (UNFORMATTED): 190:41-42

(3) If the *dtio-binding-attr* is WRITE (FORMATTED): 191:2-3

(4) If the *dtio-binding-attr* is WRITE (UNFORMATTED): 191:18-19

[Editor: Delete.] 191:30

[Editor: Delete.] 246:19-22

[Editor: Delete.] 249:18-20

[Editor: Delete.] 251:11-19

14.1.2.4.2$\frac{3}{4}$ Resolving derived-type input/output procedure references 346:22+

The *effective set of input/output procedures* for a type and *dtio-binding-attr* is the set of procedures inherited for that *dtio-binding-attr* from the parent of the type, minus the overridden ones, plus the ones declared in the type. Each procedure in an effective set has a corresponding one in each effective set for each extension type – either the same procedure or one that overrides it. Each effective set is a generic interface.

A derived-type input/output procedure for one of the four kinds of data transfer specified in 9.5.4.4.3 and a particular type of list item is selected as follows:

(1) At most one procedure is selected from the effective set of procedures for the *dtio-binding-attr* and the declared type of the list item, according to the generic resolution rules (14.1.2.4.1).

(2) If a procedure is selected in step (1), the reference is to the procedure from the effective set, for the *dtio-binding-attr* and the dynamic type of the list item, that corresponds to the procedure selected in step (1). Otherwise, intrinsic input/output is used.

If the reference is to a deferred binding, an error condition occurs.

Subject: Define "component order" term, issues 17-19 and 211, more work on constructors
From: Van Snyder
References: 00-148, 00-152

1 Introduction

In paper 00-148, Malcolm addressed issues 17-19 and 211. In paper 00-152, I addressed the definition of the "component order" term. This paper combines those two papers.

Concerning issues 17-19 and 211, Malcolm wrote in paper 00-148:

> Issue 17 says
>
> > "... I'm bothered by having a component name that isn't the name of a component. Perhaps we should use a different terminology such as subobject name ...."
>
> I concur.
>
> Issue 18 says
>
> > "Should the above not be a constraint? Fix up Grandparents."
>
> The answer to the question is "No," but it ought to be part of our scoping rules (which do have similar status as constraints in requiring violation to be diagnosed).
>
> I concur with the second commandment.
>
> Issue 19 says
>
> > "but the name ... is not a component ...."
>
> ok, ok already
>
> Issue 211 says
>
> > " 'flattened form' is used ... but ... nowhere defined"
>
> I concur.

Paper 00-148 introduced the term "subobject name." This paper instead expands on the definition of "subobject", which is defined only superficially at [16:23-28]. The term "subobject name" then follows from the term "subobject."

Section **4.5.6 Construction of derived-type values** doesn't work for extended types. Section **4.5.3.1 Inheritance** defines the order of components of an extended type, for purposes of derived-type value construction and intrinsic input/output, but doesn't define the term.

This paper defines the term "subobject order" for nonextensible, base and extended types, and uses the term for value construction and intrinsic input/output.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by +

indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

Ultimately, a nonextensible or base derived type is resolved into **ultimate components** that are either of intrinsic type or have the ALLOCATABLE or POINTER attribute. An extended type may be resolved into **ultimate subobjects** (4.5.3.1) if subobjects of the parent subobject are to be included, or ultimate components if subobjects of the parent subobject are not to be included. 41:21-22

[Editor: Delete "For purposes..." to the end of the paragraph.] 53:15-18

[Define a term for the parent subobject.] 53:22-24
An extended type has a **parent subobject** with the type and type parameters of the parent type, consisting of all of the subobjects inherited from the parent type. The name of the parent subobject is the parent type name.

[Editor: Delete issue 17. We no longer call the name of the parent subobject the "component name."] 53:25-34

[Editor: Replace "subobject denoted by the parent type name" with "parent subobject name". (Improve readability by using the newly coined term.)] 53:35

[Editor: Insert a new paragraph. Add this instance of "subobject" to the index.] 53:37+
The **subobjects** of a nonextensible type or of a base type are its components. The subobjects of an extended type are the parent subobject, subobjects of the parent subobject, and the additional components declared, if any.

The **ultimate subobjects** of a nonextensible type or of a base type are its ultimate components. The ultimate subobjects of an extended type are the ultimate subobjects of the parent subobject, and the ultimate subobjects of additional components declared, if any.

This extends the definition of the term "subobject" and thereby defines the term "subobject name." We use "subobject" instead of "component" when we want to include parent subobject(s). Note that it is defined recursively so that "grandparent" subobjects are included.	*Note to J3*

[Editor: Replace "have neither" with "not have". Replace "accessible component" with "accessible subobject". Delete "nor ... type". Make the whole thing a note. (Use our new terminology; make it a note because it will be covered by the scoping rules in section 14.)] 53:38-40

[Editor: Delete issue 18.] 53:41-43

4.5.3$\frac{1}{2}$ Subobject order 55:0+

[Editor: Insert "subobject order" into the index.]

The **subobject order** of the subobjects of a derived type is the subobject order of the parent subobject, if the type is an extended type and the parent type has subobjects, followed by the order of the declarations of components declared in the derived type definition.

The subobject order of the ultimate subobjects of a derived type is the order of the ultimate subobjects of the parent subobject, if the type is an extended type and the parent type has subobjects, followed by the order of the declarations of components that are of intrinsic type, and the ultimate subobjects that result from declarations of components of the derived type, taken in the order the declarations appear in the derived type definition.

The structure constructor for any derived type may be in **flattened form**, in which values may be provided for subobjects inherited from the parent type, if any. The structure constructor for an extended type may be in **nested form**, which allows providing a single value for the parent 55:29+

subobject.

Constraint: The type name and all subobjects of the type shall be accessible in the scoping unit containing the structure constructor. 55:32-56:4

Constraint: In the flattened form, there shall be at most one *component-spec* corresponding to each subobject of the type other than the parent subobject and no *component-spec* corresponding to the parent subobject. In the nested form, there shall be at most one *component-spec* corresponding to the parent subobject, and at most one *component-spec* corresponding to each component declared for the extended type.

Constraint: In the flattened form, there shall be exactly one *component-spec* corresponding to each subobject of the type, other than the parent subobject, that does not have default initialization. In the nested form, there shall be exactly one *component-spec* corresponding to the parent subobject of the type, and exactly one *component-spec* corresponding to each component declared for the extended type that does not have default initialization.

Constraint: The *keyword* = may be omitted from a *component-spec* only if the *keyword* = has been omitted from each preceeding *component-spec* in the constructor.

Constraint: In the flattened form, each *keyword* shall be the name of a subobject of the type. In the nested form, each *keyword* shall be the name of a component declared for the extended type, or the name of the parent subobject.

If the first *component-spec* has no keyword and the type of the *expr* is the same as the parent type, or if there is a *component-spec* with a keyword that is the same as the parent subobject name, the constructor is in nested form. Otherwise, the constructor is in flattened form.

In the nested form, in the absence of a component name keyword, the first *expr* is assigned to the parent subobject, the second *expr* is assigned to the first component declared in the derived type definition, and each subsequent *expr* is assigned to the sequentially corresponding component declared in the derived type definition.

In the flattened form, in the absence of a component name keyword, each *expr* is assigned to the corresponding subobject of the type, with the subobjects taken in subobject order ($4.5.3\frac{1}{2}$).

If the keyword is the same as the parent subobject name, the *expr* is assigned to the parent subobject; otherwise the *expr* is assigned to the subobject named by the keyword.

[Note to Editor: This includes deleting issues 19 and 211.] 56:7-20

The value that corresponds to the parent subobject is assigned to the parent subobject using intrinsic assignment.

For nonpointer components, the corresponding value is assigned to the corresponding subobject using intrinsic assignment (7.5.1.4).

Note to J3

The previous semantics were "converted according to the rules of intrinsic assignment to a value that has the same type and type parameters as the corresponding component. The shape of the expression shall correspond to the shape of the component." Since this didn't say it did intrinsic assignment, there's some question how it handles pointer and allocatable components of a derived type component value. The revision clarifies this, and also allows a scalar *expr* to be assigned to an array component.

For pointer components, the corresponding *expr* shall evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement (7.5.2), and it is assigned to the component using pointer assignment.

[Editor: Delete.] 57:1-3

[Editor: Replace "component" with "subobject". (This now includes subobjects inherited from the parent type in the case of objects of extended type.)] 87:41

[Editor: Replace "component" with "subobject". (This now includes subobjects inherited from the parent type in the case of objects of extended type.)] 89:3

[Editor: Replace "component" with "subobject". (This now includes subobjects inherited from the parent type in the case of objects of extended type.)] 91:10

[Editor: Replace "name of a component" with "subobject name". (Make parent subobjects usable).] 96:37

[Editor: Replace "components" with "subobjects". (This now includes subobjects inherited from the parent type in the case of objects of extended type.)] 103:44

[Editor: Replace "component ultimately in the object" with "ultimate subobject". (This now includes subobjects inherited from the parent type in the case of objects of extended type.)] 183:29-30

[Editor: Replace "component" with "subobject" twice. (This now includes subobjects inherited from the parent type in the case of objects of extended type.)] 183:34

[Editor: Replace "in the same ... unless" by "in the subobject order ($4.5.3\frac{1}{2}$) of the ultimate subobjects unless." 183:38-39

[Editor: Replace "components ... comprise" by "effective items (9.5.2) that result from expanding".] 188:44

[Editor: Replace "components, and binding names" with "bindings, and named subobjects". (Move scoping requirements from section 4 to section 14).] 342:5

---

ultimate subobject (4.5.3): For a *derived type* or a *structure*, a *subobject* that is of *intrinsic type*, has the ALLOCATABLE attribute, or has the POINTER attribute, or an *ultimate subobject* of a subobject that is of *derived type* and does not have the ALLOCATABLE attribute or the POINTER attribute. 407:22+

[Editor: Insert a new paragraph:] 416:24+

A subobject of a nonextensible type or of a base type is the same as a component. A subobject of an extended type is the parent subobject, a subobject of the parent type, or a component of the extended type. The distinction between an ultimate component and an ultimate subobject is that an ultimate subobject might arise from the parent subobject, whereas an ultimate component cannot. Consider the following example:

```
TYPE, EXTENSIBLE :: POINT
  REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: PERSON_POINT
  TYPE(PERSON) :: WHO
END TYPE COLOR_POINT
```

The only component of PERSON_POINT is WHO. The subobjects of PERSON_POINT are X, Y, POINT and WHO. The ultimate subobjects of PERSON_POINT are X, Y, NAME and AGE.

Subject: A problem I had in converting Fortran 77 to Fortran 95
From: Van Snyder

1 Introduction

I was given 463 files of Fortran 77 external procedures to convert to Fortran 95 module procedures. This was a library of mathematical software, without much inter-procedure interaction, so I wouldn't need to insert many USE statements. I thought I would have an easy time if I made a module for each procedure:

```
module xyz_module
contains
  include 'xyz.f'
end module xyz_module
```

and then added a few USE statements.

Unfortunately, I stumbled over the constraints at [267:9-10] and [269:5-6], that require END FUNCTION and END SUBROUTINE instead of simply END for module procedures.

A simple perl script could have fixed the END statements, but then I would have two files to maintain.

When I first saw these constraints, probably 1n 1988, I thought they were a good idea. Now, I'm having second thoughts. Can we remove the "or module" parts? It wouldn't invalidate any existing program.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Delete "or module".] 267:9-10

[Editor: Delete "or module".] 269:5-6

Subject: Issue 266
From: Van Snyder

1 Introduction

Issue 266 points out that "polymorphic objects" are defined, but in several instances we refer at least indirectly to polymorphic entities. Polymorphism should be defined by reference to entities, not objects.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Replace "objects" by "entities" and "object" by "entity".]	69:23-24
[Editor: Delete Issue 266.]	69:28-37
[Editor: Replace "objects" by "entities". Don't replace "unlimited polymorphic object" by "unlimited polymorphic entity" – I don't think we have any of the latter.]	69:42
[Editor: Replace "object" by "entity" and "objects" by "entities" throughout – several times for each.]	69:43-45

Subject: Part of issue 237
From: Van Snyder

1 Introduction

In issue 237, the editor remarks that COMPATIBLE rounding is exactly specified, but NEAREST rounding is not. This paper attempts to define the NEAREST rounding mode as precisely as is the COMPATIBLE rounding mode.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Replace "correspond ... IEEE standard" by "be the closer of the two nearest representable values, or the even value if halfway between them."] 224:39-40

[Editor: Replace "On processors ... the" by "The".] 225:2

[Editor: Delete the part of issue 237 that this paper addresses.] 225:8-15

Subject: Issue 258
From: Van Snyder

1 Introduction

In issue 258, the editor remarked that several sentences concerning program arguments are confusing and misplaced. This paper rewords them (and a few others that were more clumsy or less precise than necessary), and re-arranges some things to be clearer.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[(State in terms of assignment, so that trailing blanks get filled automatically):] 236:22-23
The processor shall assign a representation of the entire command that invoked the program to the command program argument.

[Editor: Replace "The argument ... command name" by the following (state in terms of assignment, so that trailing blanks get filled automatically):] 236:27-28
The processor shall assign a representation of the command name and the command arguments to the argument text program argument.

[Editor: Insert a new paragraph:] 236:29+
The processor shall insure that the length of the argument text program argument is not less than the maximum value of any element of the argument length program argument.

[Editor: Delete.] 236:36-40

[Editor: Delete issue 258.] 237:6-19

Subject: More work on extensible derived type definitions
From: Van Snyder
References: 00-175 00-180

1 Introduction

Malcolm says this paper is entirely unnecessary – it's covered by the scoping rules of section 14.1.2, which have constraint status, at least if the parent subobject material is sorted out, as 00-180 attempts to do.

This paper depends on some re-arrangement of text resulting from paper 00-175.

There is no constraint that a component name cannot appear more than once within a single derived type definition. Maybe the constraint at [64:16] covers that. Nonetheless, we need constraints against duplicating the parent type name, against duplicating the names of any components inherited from the parent type, against duplicating a type parameter name, and against duplicating a procedure binding.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

42:10+ Constraint: No *type-param-name* shall appear more than once in the *type-param-name-list.*
Constraint: No *type-param-name* shall be the same as the parent type name, or any component name, type parameter name, or procedure binding name inherited (4.5.3.1) from the parent type.

42:44- [Editor: The next two constraints apply to the syntax rule for *component-decl*, which will be moved to be above this point by 00-175.]

Constraint: No *component-name* shall appear more than once in all of the *component-decl-lists* within a single derived type.
Constraint: No *component-name* shall be the same as the parent type name, or any *type-param-name*, or any component name, type parameter name, or procedure binding name inherited (4.5.3.1) from the parent type.

44:18+ Constraint: No *proc-binding-name* shall appear more than once in all of the *proc-bindings* within a single derived type definition.
Constraint: No *proc-binding* shall be the same as the parent type name, or any *type-param-name*, or any component name or type parameter name inherited (4.5.3.1) from the parent type.

Subject: Semantics of the select kind construct are not described, and it appears to be a mess to use
From: Van Snyder
References: 98-179, 00-179, 00-195

1 Introduction

The select kind construct, apparently intended to be used within a derived type definition to select different specific procedures to invoke using an object of derived type, depending on the kind parameters, is not described further than providing its syntax. In particular, the relation between select kind, inheritance, and procedure overriding is not described.

Furthermore, if I understand it correctly, it is quite cumbersome to use. Suppose one has a type with three kind parameters, and one anticipates three values for each of those parameters. If one procedure is needed for each combination of kind type parameter values, this results in a requirement to bind 27 procedures to the type. It appears to require 92 statements to do so, using the select kind construct: Three nested select kind constructs are needed. The inner ones needs 8 statements each – the SELECT CASE and END SELECT statements, 3 CASE statements, and 3 procedure declaration statements. Each middle one encloses three of these, and adds five more statements, for a total of 29 statements per middle level case. The outer one has three middle ones, and adds five more statements, for a total of 92 statements. The proposal here would allow to use one statement – albeit perhaps using more than one line, but not 92 lines.

As I understand it, this is a very clumsy explicit replacement for the automatic generic resolution mechanism. (Actually, the intent is to specify how to generate dispatch tables, but the generic mechanism could do that more clearly.)

I propose in this paper to replace the select kind construct with the already-developed generic resolution mechanism.

This strategy has a simple extension to type-bound defined assignment, type-bound defined operators, type-bound derived-type input/output procedures (see 00-179), and type-bound final procedures (see 00-195).

2 Specifications

Several specific procedures may be bound to a type by using one binding name. The specific procedures bound to (not inherited into) a single type-bound procedure name shall be distinguishable according to the rules for unambiguous generic procedure reference (14.1.2.3).

The PASS_OBJ declaration applies to the binding name, and thereby to all of the specific procedures bound to the type, and all of its extensions, by that name, so we don't need to worry about the case that a binding name has PASS_OBJ in the parent type but not in the type being declared, or vice-versa.

The rules for overriding are not much more difficult than in the case of nongeneric type-bound procedures. We don't have an explanation for the semantics of the select kind construct, but I don't think it will be similer than this: If a specific procedure to be bound to a type by a particular binding name is not distinguishable from one bound to the parent by the same name, by using the rules of section 14.1.2.3, it overrides the one inherited from the parent. Otherwise, it

extends the set of procedures accessible by applying the generic procedure resolution mechanism to the binding name.

Now consider procedure invocation. Define the *effective set of procedures* for a type and binding name to be the set of procedures inherited for that binding name from the parent of the type, minus the overridden ones, plus the ones declared in the type. Each procedure in an effective set has a corresponding one in each effective set for each extension type – either the same procedure or one that overrides it. First, one procedure is selected from the effective set of procedures for the declared type of the invoking object and specified binding name, according to the generic resolution rules. Then the corresponding procedure from the effective set for the dynamic type of the invoking object and the same binding name is invoked. From an implementors point of view, there is a separate dispatch table for each distinct generic resolution of a binding name.

3 Syntax

There are (at least) two syntaxes to specify generic type-bound procedures:

1. Specify all procedure bindings by using the PROCEDURE statement. If several bindings have the same binding name, they create a generic set. This has the advantage of using only one statement, but the disadvantage of not noticing that overriding was intended instead of generic extension.

2. Specify non-generic procedure bindings by using the PROCEDURE statement, and generic bindings by using a new GENERIC statement. This has the disadvantage of requiring a new statement, and the advantage that the processor can detect one case in which one mistakenly extends the generic set instead of overriding a non-generic binding – the case when the name is already non-generic.

3.1 Syntax – first option

The *proc-binding* is extended to

R440 *proc-binding* **is** PROCEDURE[(*proc-interface-name*)] ■
■ [[, *binding-attr-list*] ::] *binding-name* ■
■ => NULL()
or PROCEDURE [[, *binding-attr-list*] ::] ■
■ *binding-name* => *procedure-name-list*

A *binding-name* specified in a PROCEDURE statement may be the same as the binding name specified in another PROCEDURE statement, having the same effect as if the *procedure-name-lists* were combined in a single statement.

3.2 Syntax – second option

The PROCEDURE statement is unchanged, and the *proc-binding* is extended to include

R440 *proc-binding* **is** <as at present>
or GENERIC[(*proc-interface-name*)] ■
■ [[, *binding-attr-list*] ::] *binding-name* ■
■ => NULL()

or GENERIC [[, *binding-attr-list*] ::] ■
■ *binding-name* => *procedure-name-list*

A *binding-name* specified in a PROCEDURE statement shall not be the same as any other binding name specified within the same derived type definition, no matter whether specified in a PROCEDURE or GENERIC statement; if it is the same as an inherited one, the present overriding rules apply – no extension of a generic set is permitted. A binding name specified in a GENERIC statement may be the same as the binding name specified in another GENERIC statement, having the same effect as if the *procedure-name-lists* were combined in a single statement.

3.3 Straw vote

Straw Vote

(a) Use the PROCEDURE statement to specify all type-bound procedure bindings,

(b) Use the PROCEDURE statement to specify non-generic type-bound procedure bindings, and the GENERIC statement to specify generic type-bound procedure bindings, or

(c) Don't do this at all. Try to make the select kind construct work instead. I dare you to try.

4 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text. – first option

There are additional edits in section 6 that apply to both options.

R440 *proc-binding* **is** PROCEDURE(*proc-interface-name*) ■ 44:17-20
■ [[, *binding-attr-list*] ::] *binding-name* ■
■ => NULL()
or PROCEDURE [[, *binding-attr-list*] ::] ■
■ *binding-name* => *binding-list*

Constraint: The binding name shall not be the same as a binding name in the parent type that is declared to be NON_OVERRIDABLE.

Constraint: If an *access-spec* is specified for a *binding-name*, the same *access-spec* shall be specified for every PROCEDURE statement that specifies the same *binding-name* within the type definition. 44:31+

The same binding name may be used in several procedure binding statements within a single type definition. The effect is as if all of the NULL() bindings were specified by NULL(*procedure-pointer-name*) with *procedure-pointer-name* specifying a procedure pointer with the same interface as the *proc-interface-name*, and then all the bindings were specified by a single statement. 49:30+

A procedure binding declared within a derived type definition overrides one inherited from the parent type if: 54:11-16

(1) The binding declared in the type has the same binding name as one inherited from the

parent type, and

(2) the specific or deferred procedure to be bound to the type by a particular binding name is not distinguishable, by using the rules of section 14.1.2.3, from one inherited from the parent and bound to the same binding name.

Otherwise, it extends the set of procedures accessible by applying the generic procedure resolution mechanism (14.1.2.4.2$\frac{1}{2}$) to the binding name. If a binding overrides one inherited from the parent, it and the inherited one shall match in the following ways:

5 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text. – second option

There are additional edits in section 6 that apply to both options.

R440 *proc-binding* **is** PROCEDURE(*proc-interface-name*) ■ 44:17-20
■ [[, *binding-attr-list*] ::] *binding-name* ■
■ => NULL()
or PROCEDURE [[, *binding-attr-list*] ::] ■
■ *binding-name* => *binding*
or GENERIC(*proc-interface-name*) ■
■[[, *binding-attr-list*] ::] *binding-name* ■
■=> NULL()
or GENERIC [[, *binding-attr-list*] ::] ■
■ *binding-name* => *binding-list*

Constraint: The binding name shall not be the same as a binding name in the parent type that is declared to be NON_OVERRIDABLE.

Constraint: If a binding name is inherited (4.5.3.2) from the parent type, then the binding name inherited from the parent type and the one being declared shall both be declared with GENERIC or both be declared with PROCEDURE.

Constraint: If an *access-spec* is specified for a *binding-name*, the same *access-spec* shall be specified for every GENERIC statement that specifies the same *binding-name* within the type definition. 44:31+

The same binding name may be used in several GENERIC procedure binding statements within a single type definition. The effect is as if all of the NULL() bindings were specified by NULL(*procedure-pointer-name*) with *procedure-pointer-name* specifying a procedure pointer with the same interface as the *proc-interface-name*, and then all the bindings were specified by a single statement. 49:30+

A procedure binding declared within a derived type definition overrides one inherited from the parent type if: 54:11-16

(1) The binding declared in the type has the same binding name as one inherited from the parent type, and

(2) it is declared using GENERIC and the specific or deferred procedure to be bound to the type by a particular binding name is not distinguishable, by using the rules of section 14.1.2.3, from one inherited from the parent and bound to the same binding name.

Otherwise, it extends the set of procedures accessible by applying the generic procedure resolution mechanism (14.1.2.4.2$\frac{1}{2}$) to the binding name. If a binding overrides one inherited from the parent, it and the inherited one shall match in the following ways:

6 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.– both options

The following edits are needed to implement generic type bound procedures, no matter what syntax is chosen.

---

proc-binding
[*proc-binding*] ... — 44:12-13

[Editor: Delete.] — 44:15-16

Constraint: If the binding name is the same as one inherited from the parent type, PASS_OBJ shall be specified if and only if it is specified for the binding of the same name in the parent type. — 44:31+

Constraint: If PASS_OBJ is specified for a binding name in one procedure binding within the derived type declaration, it shall be specified in all procedure bindings for that binding name within the derived type declaration.

Constraint: If NON_OVERRIDABLE is specified for a binding name in one procedure binding within the derived type declaration, it shall be specified in all procedure bindings for that binding name within the derived type declaration.

---

or NULL(*procedure-name*)
or NULL(*procedure-pointer-name*) — 44:45

[Editor: Replace "procedure that has" by "procedure. The *procedure-pointer-name* shall be the name of an accessible procedure pointer. The procedure or procedure pointer shall have". After the second "procedure" insert "or procedure pointer".] — 44:47

[Editor: Delete.] — 45:4-13

[Editor: Replace "deferred" with "a **deferred procedure binding**".] — 49:26

---

may override (4.5.3.2) the inherited deferred binding with another deferred binding. — 49:29

[Editor: Delete "in that interface block".] — 345:7

[Editor: Delete "in that interface block".] — 345:12-13

[Editor: Add a new section. The term *effective set of procedures* is defined here, but not used anywhere other than in this section. I've set it in italic instead of bold face, with the intention that it's not worth putting in the index and glossary. If you want to set it in bold face and put — 346:22+

it in the index, that's fine, too. If you set it in bold face, do I owe you a glossary entry?]

14.1.2.4.2½ Resolving type bound procedure references

The *effective set of procedures* for a type and binding name is the set of procedures inherited for that binding name from the parent of the type, minus the overridden ones, plus the ones declared in the type. Each procedure in an effective set has a corresponding one in each effective set for each extension type – either the same procedure or one that overrides it. For purposes of generic resolution, the passed-object dummy argument (4.5.1) of a procedure inherited from the parent type is considered to have the extended type into which it is inherited. Each effective set of procedures is a generic interface.

If a type-bound procedure is specified by *data-ref % binding-name* in a function reference or call statement:

(1) One procedure is selected from the effective set of procedures for the *binding-name* and the declared type of the *data-ref*, according to the generic resolution rules (14.1.2.4.1).

(2) The reference is to the procedure from the effective set, for the *binding-name* and the dynamic type of *data-ref*, that corresponds to the procedure selected in step (1).

If the reference is to a deferred binding, an error condition occurs.

---

deferred procedure binding (4.5.1.5): a *type-bound procedure* binding that specifies the NULL() intrinsic. A deferred procedure binding shall not be invoked. 400:17+

7 Straw vote about *access-spec* semantics

It is possible, by removing the constraint introduced at [44:31+] in sections 4 and 5 above, to allow some bindings to be private, and some to be public, for the same binding name. This is different from the usual rules for generic interfaces accessed from a module. Instead of the constraint would be a note:

Note 4.19½ 44:43+

It is possible for some of the bindings to a binding name to be PRIVATE and some to be PUBLIC; it is not required that all be PRIVATE or that all be PUBLIC. Within the module containing the derived type definition, all procedures bound to a type by a particular binding name are candidates for access by applying the generic resolution rules to the binding name. Without the module containing the derived type definition, only the PUBLIC procedures bound to a type by a particular binding name are candidates for access by applying the generic resolution rules to the binding name.

(a) Should the *access-spec* apply to the binding (mixed public and private), or (b) Should the *access-spec* apply to the binding name (all public or all private)?	**Straw Vote**

Subject: Discussions of INTRINSIC and EXTERNAL attributes are scattered; discussion of INTRINSIC is contradictory and repetitive
From: Van Snyder

1 Introduction

We have tried to consolidate discussion of attributes into section 5.1.2. The discussion of the INTRINSIC attribute, however, is scattered between 5.1.2.11 and 12.3.2.4 (The INTRINSIC statement), and the discussion of the EXTERNAL attribute is scattered between 5.1.2.10 and 12.3.2.2. Some of the discussion is contradictory – section 12.3.2.4 allows any intrinsic procedure named in an INTRINSIC statement to be used as an actual argument, while section 5.1.2.11 limits this set to those listed in section 13.15 and not marked with a bullet (•). Finally, some of the discussion is repeated.

This paper

(1) Consolidates discussion of the INTRINSIC attribute in section 5.1.2.11, and reduces section 12.3.2.4 to stating that the INTRINSIC statement confers the INTRINSIC attribute, and giving its syntax, and

(2) Consolidates discussion of the EXTERNAL attribute in section 5.1.2.10, and reduces section 12.3.2.2 by deleting material that is redundant and misleading.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

A dummy argument that has the EXTERNAL attribute is a dummy procedure or a dummy procedure pointer. A name that has the EXTERNAL attribute and that is not a dummy argument is the name of an external procedure, a procedure pointer, or a block data program unit. 76:39+

Note 5.18$\frac{1}{2}$

It is necessary to use an EXTERNAL statement (12.3.2.2) to specify the EXTERNAL attribute for a block data program unit; it is not possible to do so in a type declaration statement.

The **INTRINSIC attribute** confirms that a name is the specific name (13.15) or generic name (13.13, 13.14) of an intrinsic procedure. The INTRINSIC attribute allows the specific name of an intrinsic procedure that is listed in section 13.15 and not marked with a bullet (•) to be used as an actual argument (12.4). 76:41-77:2

Declaring explicitly that a generic intrinsic procedure name has the INTRINSIC attribute does not cause that name to lose its generic property.

The following constraint applies to syntax rules R504 and R1214:

Constraint: If the name of a generic intrinsic procedure is explicitly declared to have the INTRINSIC attribute, and it is also the generic name in one or more generic interfaces (12.3.2.1) accessible in the same scoping unit, the procedures in the interfaces and the specific intrinsic procedures shall all be functions or all be subroutines, and the characteristics of the specific intrinsic procedures and the procedures in the interfaces shall differ as specified in section 14.1.2.3.

The INTRINSIC attribute may also be declared by the INTRINSIC statement (12.3.2.4).

[Editor: Delete – moved to 5.1.2.10 and stated in terms of the attribute, not the statement. The present wording implies that the only way to declare a dummy procedure or external procedure is to put its name in an EXTERNAL statement. This isn't necessarily true: The EXTERNAL attribute can be specified by an interface body, or specified for a function in a type declaration statement.] 251:43-252:2

[Editor: Delete misleading note. The correct story is in 5.1.2.10, which is referenced at [251:39].] 252:11-13

[Editor: Insert "(5.1.2.11)" after "attribute" and delete "A name ..." to the end of the paragraph.] 253:44-47

[Editor: Delete.] 254:3-11

Constraint: If an actual argument is a name that is explicitly declared to have the INTRINSIC attribute, it shall not be the specific name of an intrinsic procedure that is listed in section 13.15 and marked with a bullet (•). 255:22+

Subject: Miscellaneous items
From: Van Snyder
References: 00-179

Here are several things that may or may not need attention. I don't even offer edits (well, sometimes I offer crappy ones). If they need attention, we can develop edits at the meeting, if we have time, or insert unresolved issue notes.

Shouldn't this paragraph be a constraint? 82:9-12

We find here "A user-defined derived-type input/output procedure is any procedure...." I think we do not intend to allow internal and dummy procedures, or procedure pointers. The sentence has other problems as well: It isn't enough for a procedure to have the appropriate interface; it needs to be specified in the appropriate interface block. The sentence doesn't contribute anything that's not said elsewhere in the section. Delete it. If not, at least make it consistent with the constraint at [246:30-31]. Also note that one of the proposals in paper 00-179 is to replace the interface-block-based derived-type input/output procedure specification by one based on type-bound procedures. 189:26

Everything in 11.1.2 is said elsewhere, frequently as a constraint. Can we delete section 11.1.2? 237:42-45

Not needed – it's covered by 14.1.2.3. 246:35-36

Do we need to add "accessible" after "entity," or was the intent to restrict IMPORT to work only for entities declared within the scoping unit containing the interface body? 246:39

The "otherwise" part is not true for abstract interface blocks. 247:3-4

We may want to point out in a comment that because argument B1 has assumed shape and argument B2 does not, a non-contiguous array section can be the actual argument associated with B1, but a non-contiguous array section cannot be the actual argument associated with B2. 250:4-5

It would be convenient to be able to use any accessible explicit interface to declare the interface for a procedure pointer. Could we add "**or** *procedure-name*" as an additional right-hand side for R1211? We would also need to replace "consists ... pointers" by "and specifies an explicit specific interface, the declared procedures or procedure pointers have the same explicit specific interface" at [253:7]. 252:19+

The phrase "an elemental intrinsic actual procedure may be associated with a dummy argument that is not elemental" leads one to believe that dummy arguments can be elemental. The part "that is not elemental" should be removed. Three possibilities for what to do next are (1) nothing, (2) add a parenthetic remark "(which cannot be elemental)", or (3) put in a note $12.27\frac{1}{2}$ to the effect that dummy arguments cannot be elemental. 260:9-10

We could get rid of "other than as the argument of the PRESENT intrinsic function" by making the argument of the PRESENT intrinsic function optional. 261:12-23

I think the reason for this condition is to provide bounds for the elemental-ness. If so, this condition is too strong (the dummy argument of the elemental procedure can't be optional), and not strong enough (the specified array doesn't necessarily provide the desired bounds). It should be "... unless an array of the same rank that is (1) not a dummy argument or is a present dummy argument, (2) not an unallocated allocatable array, and (3) not a disassociated pointer, is supplied as an actual argument of that elemental procedure." 261:15-17

There is at least one, and maybe two problems here. In the phrase "correspond by name to a 343:34-35

dummy argument not present" does "not present" mean "not declared," or "it has the optional attribute and there is no associated actual argument?" I think it's the former, but we do have a section with the phrase "dummy arguments not present" in its title – and it refers to the latter. The wording should be revised to avoid this confusion. In the former case, it is impossible for a nonoptional dummy argument to correspond by name to a dummy argument not present. The dummy argument that is not present clearly doesn't have a name.

The sentence "If a generic..." conflicts with, or at least belongs in [344:25-26]. 343:42-44

Not needed, because of [344:40] and the new language in 5.1.2.10 that specifies that an interface body confers the EXTERNAL attribute. Perhaps [344:40] should be re-worded "(d) if there is an explicit specification of the EXTERNAL attribute (5.1.2.10) in that scoping unit". 344:35

Subject: One more try at a READONLY attribute
From: Van Snyder
References: 00-169 00-192

1 Introduction

The people I work with, who pay the bill for me to participate in J3 meetings, have asked me again about a READONLY attribute for module variables. I mentioned that it had foundered on the name: "Hmmm, READONLY... does that mean it can only appear in a read statement?"

Several other names were suggested for the attribute: LIMITED, SEMIPRIVATE (with and without an underscore) and PROTECTED.

I'll use LIMITED here, because it's the shortest one. If the proposed specification is accepted, we can have a straw vote on the spelling.

I propose here that we add an attribute, however spelled, that a named variable cannot be changed, and the pointer association status of a pointer cannot be changed, in scoping units that access the variable by use association. By stretching our imaginations a little bit, we can put it under the aegis of work plan item R4, which I thought had been changed to be something like "Improving modules so that it's easier to use modules to implement new abstract data types efficiently," but the work plan (00-010) still says "Interval Arithmetic."

2 Specification

Add an attribute and statement that specifies that a named variable cannot be changed, and the pointer association status of a pointer cannot be changed, in scoping units that access the variable by use association.

3 Syntax

Except for spelling, the syntax is obvious: An attribute and statement, spelled with the same keyword. **Straw vote**: (1) LIMITED, (2) PROTECTED, (3) SEMIPRIVATE or SEMI_PRIVATE, (4) other (and another obvious straw vote if (3) wins).

4 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

or *limited-stmt*	10:49+
or LIMITED	42:27+
Constraint: If PRIVATE appears, LIMITED shall not appear.	42:30+

or LIMITED 42:40+

or LIMITED 64:1+

Constraint: The LIMITED attribute shall be specified only in the specification part of a module. 65:12+
Constraint: The LIMITED attribute shall be specified only for a named variable.
Constraint: If the LIMITED attribute is specified, the EXTERNAL, INTRINSIC, PARAMETER, PRIVATE or PUBLIC attribute shall not be specified.
Constraint: The LIMITED attribute shall not be specified for an object that is in a common block.

5.1.2.9½ LIMITED attribute 76:21+

The **LIMITED attribute** specifies that a named variable or structure component shall not appear in a variable definition context (14.7.7) in any scoping unit that accesses it by use association. If it has the POINTER attribute its association status shall not be changed or become undefined in any scoping unit that accesses it by use association. A named variable with the LIMITED attribute may be referenced in a scoping unit that accesses it by use association, even if the default accessibility is PRIVATE. A structure component with the LIMITED attribute may be referenced in a scoping unit that accesses it by use association, even if the default accessibility for components of the type of the object is PRIVATE.

5.2.9½ LIMITED statement 82:23+

R533½ *limited-stmt* **is** LIMITED [::] *object-name-list*

The **LIMITED statement** specifies the LIMITED attribute for a list of objects.

[Editor: Insert ", a LIMITED statement" before "or" – but not if section 11.1.2 is deleted as recommended in 00-192. 237:44

[Editor: Insert ", the LIMITED statement (5.2.9½)" after "(5.2.3)".] 239:17

[Editor: Insert ", 5.1.2.9½" after "5.1.2.2", and replace "statement" by "and LIMITED statements".] 239:18

[Editor: Replace "or PRIVATE" by ", PRIVATE or LIMITED".] 240:40

[Editor: Before "If" insert "If the identifier appears in a LIMITED statement it causes the object accessible by use association to be a limited object of that module.] 241:3

[Editor: Replace "either a PUBLIC or PRIVATE" by "a PUBLIC, PRIVATE or LIMITED".] 241:4

[Editor: Replace "or PUBLIC" by ", PUBLIC or LIMITED" – but not if the "PRIVATE or PUBLIC" part is removed as advocated in 00-169.] 242:2

Subject: FINAL procedures as type-bound procedures
From: Van Snyder
References: 99-108, 00-138 00-170 00-186

1 Introduction

This paper is based on 00-138, which was available but not discussed at meeting 152. The syntax is slightly different from what was adopted for final procedures in 99-108, and slightly different from what was proposed in 00-138. There is more work to be done for final procedures, especially specifying the order in which objects cease to exist, and therefore the order in which their final procedures are executed.

This paper depends on paper 00-186. It should be processed, if at all, after that paper passes, or another one that specifies how kind type parameters interact with type-bound procedure invocation passes.

2 Edits

Edits refer to 00-007. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.for finalization

---

44:18+

or PROCEDURE (*proc-interface-name*), ■
■FINAL => NULL()
or PROCEDURE, FINAL => *final-binding-list*

44:25

[Editor: Add to the constraint:]
If *proc-interface-name* and FINAL are both specified, the interface shall specify a subroutine that has one dummy argument with a declared type of *type-name* and that is polymorphic if and only if *type-name* is extensible. This argument shall not have INTENT(OUT), nor have the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, VALUE or VOLATILE attribute. If the dummy argument is an array it shall have assumed shape. All nonkind parameters of the dummy argument shall be assumed.

Change "VALUE" to "INTENT(VALUE)" and put it before OPTIONAL if paper 00-170 passes.	*Editor*

44:31+

final-binding **is** *procedure-name*
or NULL(*procedure-name*)
or NULL(*procedure-pointer-name*)

Constraint: The *procedure-name* shall be the name of an accessible module procedure or external procedure. The *procedure-pointer-name* shall be the name of an accessible procedure pointer. The procedure or procedure pointer shall be a subroutine with an explicit interface having one dummy argument with a declared type of *type-name* and that is polymorphic if and only if *type-name* is extensible. This argument shall not have the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL,

INTENT(OUT), POINTER, VALUE or VOLATILE attribute. If the dummy argument is an array it shall have assumed shape. All nonkind parameters of the dummy argument shall be assumed.

Change "VALUE" to "INTENT(VALUE)" and put it before OPTIONAL if paper 00-170 passes.	*Editor*

Constraint: If several subroutines are bound to the type with *binding-attr* FINAL, they shall be distinguished according to the rules for unambiguous procedure references (14.1.2.3).

4.5.1.5.1 Final subroutine 49:30+

A procedure binding that specifies FINAL is a **final subroutine** for objects of the type. The set of final subroutines that are bound to the type or that are inherited (4.5.3.1) from the parent type and not overriden (4.5.3.2) is a generic interface.

If any final subroutines are specified for a type and set of kind type parameters, at least one of them shall have a scalar dummy argument.

A final subroutine may be elemental.

When any object is deallocated (6.3.3, 6.3.3.1) or becomes undefined by the events specified by items (3) or (13)(c) in 14.7.6, if a final subroutine is selected as specified in 14.1.2.4.2$\frac{2}{3}$, it is invoked with the object as its actual argument. If the subroutine causes other objects of the same type and kind type parameters to be deallocated or to become undefined by the events specified by items (3) or (13)(c) in 14.7.6, it shall be recursive.

Immediately following execution of a final subroutine, if it overrides (4.5.3.2) one, the overridden final subroutine is invoked, with the object as its actual argument. This process is repeated until no further final subroutine is available.

Immediately following this process, the object becomes deallocated or undefined.

If a procedure binding that specifies FINAL (4.5.1.6) cannot be distinguished from one inherited (4.5.3.1) from the parent type according to the rules for unambiguous procedure references (14.1.2.3), it overrides that binding. 55:0-

---

346:22++ (after material inserted at this point by 00-186)

14.1.2.4.2$\frac{2}{3}$ Resolving final procedure references

The *effective set of final subroutines* for a type is the set of final subroutines inherited from the parent of the type, minus the overridden ones, plus the ones declared in the type. Each subroutine in an effective set has a corresponding one in each effective set for each extension type – either the same subroutine or one that overrides it. Each effective set of final subroutines is a generic interface.

A final subroutine for an object is selected by:

(1) At most one subroutine is selected from the effective set of final subroutines for the declared type of the object, according to the generic resolution rules (14.1.2.4.1).

(2) If a subroutine is selected in step (1), the reference is to the subroutine from the effective set, for the dynamic type of the object, that corresponds to the subroutine selected in step (1).

If the reference is to a deferred binding, an error condition occurs.

Subject: Edits to implement decisions in 00-155 concerning ALLOCATABLE
From: Van Snyder
References: 00-155

1 Introduction

In paper 00-155, Malcolm provided alternative syntaxes to implement the capability to specify the type of object to be allocated by reference to another object. There was a straw vote, and the syntax "SOURCE = *source-variable*" won. This paper provides edits, as outlined in 00-155 but updated to reflect the result of the straw vote and to refer to 00-007r1.

2 Edits

Edits refer to 00-007r1. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

---

or SOURCE = *source-variable* 102:15+

R631a *source-variable* **is** *variable* 102:23+

Constraint: If SOURCE= is specified, *type-spec* shall not be specified, *allocation-list* shall contain only one *allocation*, and *allocate-object* shall be type compatible with *source-variable*. 102:38+

Constraint: The *source-variable* shall be polymorphic and have the same rank as the *allocate-object*.

Constraint: Corresponding kind type parameters of *allocate-object* and *source-variable* shall have the same values.

[Editor: After ";" insert "if a *source-variable* is specified, it allocates an object whose dynamic type and type parameters are the same as those of the *source-variable*;"] 103:6

If SOURCE= is present, *source-variable* shall have the same shape as *allocate-object*. If the value of a nondeferred nonkind type parameter of *allocate-object* is different from the value of the corresponding type parameter of *source-variable*, an error condition occurs. If the allocation is successful, *source-variable* is then assigned to *allocate-object* as if by intrinsic assignment for objects whose declared type is the dynamic type of *source-variable*. 103:25+

ISO/IEC JTC1/SC22/WG5**J3/00-197**

ISO IEC TECHNICAL REPORT

ISO/IEC JTC1 PDTR XX.XX.XX

Enhanced Module Facilities

in

Fortran

An extension to IS 1539-1:1997

13 March 2000

THIS PAGE TO BE REPLACED BY ISO-CS

Contents

Foreword

[General part to be provided by ISO CS]

This technical report specifies an extension to the module program unit facilities of the programming language Fortran. Fortran is specified by the international standard ISO/IEC 1539-1. This document has been prepared by ISO/IEC JTC1/SC22/WG5, the technical working group for the Fortran language.

It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran standard (ISO/IEC 1539-1) without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing commercial implementations.

0 Introduction

The module system of Fortran, as standardized by ISO/IEC 1539-1, while adequate for programs of modest size, has shortcomings that become evident when used for large programs, or programs having large modules. The primary cause of these shortcomings is that modules are monolithic.

This technical report to extends the module facility of Fortran so that program developers can encapsulate the implementation details of module procedures in zero or more **submodules**, that are separate from but dependent on the module in which the interfaces of their procedures are defined. If a module or submodule has submodules, it is the **parent** of those submodules.

The facility specified by this technical report is compatible to the module facility of Fortran as standardized by ISO/IEC 1539-1.

0.1 Shortcomings of Fortran's module system

The shortcomings of the module system of Fortran, as specified by ISO/IEC 1539-1, and solutions offered by this technical report, are as follows.

0.1.1 Avoiding recompilation cascades

Once the design of a program is stable, most changes in modules occur in the implementation of those modules – in the procedures that implement the behavior of the modules and the private data they retain and share – not in the interfaces of the procedures of the modules, nor in the specification of publicly accessible types or data entities. Changes in the implementation of a module have no effect on the translation of other program units that access the changed module. The existing module facility, however, draws no structural

distinction between interface and implementation. Therefore, if one changes any part of a module, the language translation system has no alternative but to conclude that a change may have occurred that could affect other modules that access the changed module. This effect cascades into modules that access modules that access the changed module, and so on. This can cause a substantial expense to re-translate and re-certify a large program.

Using facilities specified in this technical report, implementation details of a module can be encapsulated in submodules, so that they can be changed without implying that other modules must be translated differently.

If a module is used only in the implementation of a second module, a third module accesses the second, and one changes the interface of the first module, utilities that examine the dates of files have no alternative but to conclude that a change may have occurred that could affect the translation of the third module.

Modules can be decomposed using facilities specified in this technical report so that a change in the interface of a module that is used only in a submodule has no effect on the the parent of that submodule, and therefore no effect on the translation of other modules that use the second module. Thus, compilation cascades caused by changes of interface can be shortened.

0.1.2 Packaging proprietary software

If a module as specified by the international standard ISO/IEC 1539-1 is used to package proprietary software, the source text of the module cannot be published as authoritative documentation of the interface of the module, without either exposing trade secrets, or requiring the expense of separating the implementation from the interface every time a revision is published.

Using facilities specified in this technical report, one can publish the source text of the module as authoritative documentation of its interface, while witholding publication of the source text of the submodules that contain the implementation details, and the trade secrets embodied within them.

0.1.3 Decomposing large and interconnected facilities

If an intellectual concept is large and internally interconnected, it requires a large module to implement it. Decomposing such a concept into components of tractable size using modules as specified by ISO/IEC 1539-1 may require one to convert private data to public data.

A concept can be decomposed into modules and submodules of tractable size using facilities specified in this technical report, without exposing private entities to uncontrolled use.

Decomposing a complicated intellectual concept may furthermore require circularly dependent modules. The latter is prohibited by ISO/IEC 1539-1. It is frequently the case, however, that the dependence is between the implementation of some parts of the concept and the interface of other parts. Because the module facility defined by international standard ISO/IEC 1539-1 does not distinguish between the implementation and interface, this distinction cannot be exploited to break the circular dependence. Therefore, modules that implement large intellectual concepts tend to become large, and therefore expensive to maintain reliably.

Using facilities specified in this technical report, complicated concepts can be implemented in submodules that access modules, rather than modules that access modules, thus reducing the possibility for circular dependence between modules.

0.1.4 Easier library creation

Most Fortran translator systems produce a single file of computer instructions, called an *object file*, for each module. This is easier than producing a separate object file for the specification part and for each module procedure. It is also convenient, and conserves space and time, when a program uses all or most of the procedures in each module. It is inconvenient, and results in a larger program, when only a few of the procedures in a general purpose module are needed in a particular program.

If modules are decomposed using facilities specified in this technical report, it would be easier for each program unit's author to control how module procedures are allocated among object files.

0.2 Disadvantage of using this facility

Translator systems will find it more difficult to carry out inter-procedural optimizations if the program uses the facility specified in this technical report. When translator systems become able to do inter-procedural optimization in the presence of this facility, it is likely that requesting inter-procedural optimization will cause compilation cascades in the first situation mentioned in section 0.1.1, even if this facility is used. Although one advantage of this facility would be nullified in the case when users request inter-procedural optimization, it would remain if users do not request inter-procedural optimization, and the other advantages remain in any case.

Information technology – Programming Languages – Fortran

Technical Report: Enhanced Module Facilities

1 General

1.1 Scope

This technical report specifies an extension to the module facilities of the programming language Fortran. The current Fortran language is specified by the international standard ISO/IEC 1539-1 : Fortran. The extension allows program authors to develop the implementation details of concepts in new program units, called **submodules**, that cannot be accessed directly by use association. In order to support submodules, the module facility of international standard ISO/IEC 1539-1 is changed by this technical report in such a way as to be upwardly compatible with the module facility specified by international standard ISO/IEC 1539-1.

Section 2 of this technical report contains a general and informal but precise description of the extended functionalities. Section 3 contains detailed editorial changes which if applied to the current international standard would implement the revised language specification.

1.2 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this technical report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this technical report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 1539-1 : 1997 *Information technology - Programming Languages - Fortran*

2 Requirements

The following subsections contain a general description of the extensions to the syntax and semantics of the current Fortran programming language to provide facilities for submodules.

2.1 Modules

As specified in ISO/IEC 1539-1, a module consists of a specification part and a module subprogram part.

This technical report defines a **submodule specification part**, in which only the interfaces of procedures in submodules are declared. This part is introduced by a statement of the form `SUBMODULE :: ` *submodule-name*. A submodule specification part extends from the SUBMODULE statement that introduces it to (but not including) the next CONTAINS, SUBMODULE or END MODULE statement. A **submodule procedure** is a module procedure for which the interface is specified in a submodule specification part, and the body is defined in a submodule.

A module or submodule may have any number of module subprogram parts, and any number of submodule specification parts, in any order. If several submodule specification parts have the same name, the effect is as if the specifications they contain were concatenated within a single submodule specification part. This allows one to put all module procedures into alphabetical order.

Within a submodule specification part, procedure interface declarations specify procedures in the specified subsidiary submodule that can be accessed. This interface is syntactically identical to an interface body, but semantically different in that entities of the host environment of the interface are accessible within the interface by host association. Because of this difference, a procedure interface declaration within a submodule specification part is called a **procedure interface declaration** instead of an interface body.

2.1.1 Example of a submodule specification part

```
SUBMODULE :: POINTS_A
  REAL FUNCTION POINT_DIST ( A, B )
  ! Compute the distance between the points A and B
    TYPE(POINT) :: A, B
  END FUNCTION POINT_DIST
```

The submodule specification part in the above example specifies that there is a submodule, named `POINTS_A`, and that there is a function named `POINT_DIST`, with the specified interface, that can be accessed from that submodule. If the program unit containing the submodule specification part is a module, and `POINT_DIST` is public, then `POINT_DIST` can be accessed by use association of that module.

2.2 Submodules

A **submodule** is a program unit that is dependent on and subsidiary to a module or another submodule. If a module or submodule has subsidiary submodules, it is the **parent** of those subsidiary submodules.

A submodule is introduced by a statement of the form `SUBMODULE` (*parent-name*) *submodule-name*, and terminated by a statement of the form `END SUBMODULE` *submodule-name*.

A submodule may have a specification part, zero or more submodule specification parts, and zero or more module procedure parts.

Everything is a submodule is effectively PRIVATE except for those submodule procedures that were declared to be PUBLIC in the parent module. It is not possible to access entities declared in the specification part of a submodule because a USE statement must specify a module, not a submodule. Thus, PRIVATE and PUBLIC declarations are not permitted in a submodule.

2.2.1 Completing a procedure declared in a parent module or submodule

If a procedure interface declaration appears in the parent program unit, the procedure shall be defined in the specified submodule, either within a module procedure part or a submodule specification part.

Within a module procedure part of the subsidiary submodule, the procedure body shall be introduced by a statement of the form `SUBMODULE FUNCTION` *function-name* or `SUBMODULE SUBROUTINE` *subroutine-name*, depending on the declaration in the parent program unit. The interface of the procedure shall not be repeated in the submodule.The procedure body is logically an extension of its interface declaration; it does not access its interface declaration by host assciation.

Within a submodule specification part of the subsidiary submodule, the same statement may be used to indicate that definition of the body of the procedure is deferred to a yet more subsidiary submodule. In this case, neither an interface nor body shall follow the statement. The procedure shall be defined in the submodule specified in the submodule specification part of the subsidiary submodule, either within a module procedure part or a submodule specification part. This facility may be used to place the body of a public procedure in a submodule two or more steps subsidiary to the module, so that it may share implementation-dependent data or procedures in an intermediate subsidiary submodule with procedures in different subsidiary submodules. If the procedures in the intermediately subsidiary submodule are not specified in the module, they cannot be accessed by use association, and therefore either their interfaces or bodies can be changed without affecting the translation of a program unit that accesses the module by use association.

Example of a submodule

```
SUBMODULE(POINTS) POINTS_A
  CONTAINS
    SUBMODULE FUNCTION POINT_DIST RESULT(HOW_FAR)
```

```
    ! Don't re-declare dummy arguments, or result type
      HOW_FAR = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
    END FUNCTION POINT_DIST
END SUBMODULE POINTS_A
```

Example of submodules with a deferred procedure body

```
SUBMODULE(POINTS) POINTS_A
  ! Type and data declarations shared by submodules of POINTS_A (but not
  ! accessible anywhere else:
  ...
  SUBMODULE :: SUB_POINTS_A
    SUBMODULE FUNCTION POINT_DIST
    ! No body, because it's in a SUBMODULE specification part
    ...
  ! Other submodule or contains parts
END SUBMODULE POINTS_A

SUBMODULE(POINTS_A) SUB_POINTS_A
  CONTAINS
    SUBMODULE FUNCTION POINT_DIST RESULT(HOW_FAR)
    ! Don't re-declare dummy arguments, or result type
    ! Entities in POINTS_A and POINTS can be accessed
      HOW_FAR = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
    END FUNCTION POINT_DIST
END SUBMODULE SUB_POINTS_A
```

2.3 Relation between modules and submodules

Public entities of a module, including procedure interface declarations in submodule specification parts, can be accessed by use association. Submodules contain no public entities. Public procedure interface declarations in submodule specification parts of modules imply that the procedure bodies in the specified submodules are indirectly accessible, by use association of the module.

All entities of a parent module or submodule, including private entities, declarations of interfaces to procedures implemented in different submodules, and entities accessed from a parent module or submodule by host association, are accessible within each subsidiary submodule by host association.

A procedure body in a submodule is logically a continuation of its interface in its parent program unit; it does not access its interface by host association.

3 Required editorial changes to ISO/IEC 1539-1 : 1997

The following suggested editorial changes illustrate that the extension in this technical report is not a large change to Fortran. While every effort has been made to cover all the bases, they will undoubtedly be a few additional changes necessary. Depending on the schedule of implementation of this technical report, it may also be necessary to convert the changes to refer to the 200x standard instead of the 1997 standard.

[Page and line numbers in brackets refer to ANSI/NCITS/J3 document 97-007r2.]

[10:30+] Add a new syntax rule in section **2.1 High level syntax**, after rule R213:

submodule-specification-part **is** *submodule-specification-stmt*
submodule-procedure-declaration
[*submodule-procedure-declaration*] ...

[11:35] In the second line of **2.2 Program unit concepts**, add ", a submodule" after "a module".

[11:45] In item (2) of the list in section **2.2 Program unit concepts**, replace "body" by "declaration (12.3.2)".

[186:17-34] Replace the normative text of section **11.3 Modules** (but not subsidiary sections or the notes) with the following:

A **module** contains specifications and definitions that may be accessible to other program units.

module **is** *module-stmt*
[*specification-part*]
[*procedure-part*] ...
end-module-stmt

module-stmt **is** MODULE *module-name*

procedure-part **is** *module-subprogram-part*
or *submodule-specification-part*

submodule-specification-stmt **is** SUBMODULE :: *submodule-name*

submodule-procedure-declaration **is** *procedure-interface-declaration*
or *submodule-procedure-stmt*

submodule-procedure-stmt **is** SUBMODULE FUNCTION *function-name*
or SUBMODULE SUBROUTINE *subroutine-name*

end-module-stmt **is** END MODULE [*module-name*]

Constraint: If *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* in the *module-stmt*.

Constraint: A *specification-part* in a module or submodule shall not contain a *stmt-function-stmt*, an *entry-stmt* or a *format-stmt*.
Constraint: If an object of a type for which *component-initialization* (R429) is specified appears in the *specification-part* of a module or submodule and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.
Constraint: A *module-name* shall not be the same as any other name in the program unit.
Constraint: A *submodule-name* shall not be the same as any other name in the program unit, except that two *submodule-specification-parts* may have the same name.
Constraint: The *function-name* or *subroutine-name* in a *submodule-procedure-stmt* shall be declared to be a function or subroutine, respectively, in a *submodule-procedure-declaration* in a *submodule-specification-part* of the parent module or submodule that names the submodule in which the *submodule-procedure-stmt* appears.
Constraint: A *submodule-procedure-stmt* shall not appear except within a submodule.

A module name is a global name, and shall not be the same as the name of any other program unit, external procedure, or common block in the program.

If a module has submodules ([new section] 11.3.1), it is the **parent module** of those submodules.

A *submodule-name* specified in a *submodule-specification-stmt* shall be the same as the name of exactly one submodule ([new section] 11.3.1) in the program.

Every procedure that is named in a *submodule-specification-part* and is not a dummy procedure is a submodule procedure ([new section] 12.5.2.1), and shall be declared in a *submodule-procedure-stmt*, a *submodule-function-stmt*, or a *submodule-subroutine-stmt* in the submodule specified by the *submodule-name* in the *submodule-specification-stmt*.

If the same *submodule-name* appears on more than one *submodule-specification-stmt*, the effect is as though the *submodule-specification-parts* introduced by those statements were concatenated.

[187:2+] Insert the following before the existing section **11.3.1 Module reference**, and renumber subsequent sections:

11.3.1 Submodules

A **submodule** is a program unit that is dependent on and subsidiary to its parent module or submodule. Its parent module or submodule is its host environment.

submodule **is** *submodule-stmt*
[*specification-part*]
[*procedure-part*] ...
end-submodule-stmt

submodule-stmt **is** SUBMODULE (*parent-name*) *submodule-name*

end-submodule-stmt **is** END SUBMODULE [*submodule-name*]

Constraint: The *submodule-name* in the *submodule-stmt* shall appear in a *submodule-specification-stmt* in

the module or submodule named by the *parent-name*.

Constraint: If *submodule-name* is specified in the *end-submodule-stmt*, it shall be identical to the *submodule-name* in the *submodule-stmt*.

Constraint: The *submodule-name* specified in the *submodule-stmt* shall not be the same as any other name in the program unit.

A *submodule-name* is a global name, and shall not be the same as the name of any other program unit, external procedure, or common block in the program.

If a submodule has submodules, it is the **parent submodule** of those submodules.

Note

Related submodules and their parent module or submodule stand in a tree-like hierarchical relationship one to another, with the module at the root. For each submodule, its parent module or submodule is its parent with respect to the tree, and its submodules are children with respect to the tree.

[193:25] In the first sentence of section **12.3.2 Specification of the procedure interface**, add ", submodule" after "module".

[193:27-28] In the last sentence of the first paragraph of section **12.3.2 Specification of the procedure interface**, change the first occurrence of "in an interface block" to "as a procedure interface declaration," and change the second "interface block" to "procedure interface declaration."

[193:29-33] Remove note 12.3 – it is modified and moved to section 12.3.2.1. Replace it by:

procedure-interface-declaration **is** *function-stmt*
[*specification-part*]
end-function-stmt
or *subroutine-stmt*
[*specification-part*]
end-subroutine-stmt

Constraint: A *procedure-interface-declaration* for a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, or procedure arguments.

Constraint: A *procedure-interface-declaration* shall not contain an *entry-stmt, data-stmt, format-stmt,* or *stmt-function-stmt.*

A procedure interface declaration specifies all of the procedure's characteristics.

[193:42-6] Replace the definition of *interface-body* (R1205) and the following constraint by:

interface-body **is** *procedure-interface-declaration*

Note 12.3

An interface body cannot be used to describe the interface of an internal procedure, a module procedure, of an intrinsic procedure because the interfaces of such procedures are already explicit. The name of a module procedure may, however, appear in a MODULE PROCEDURE statement in an interface block.

[194:11-12] Remove the first constraint after syntax rule R1207 (it has been moved to section 12.3.2, with revised wording).

[194:32] In the first sentence of the paragraph of text immediately before note 12.4, replace "An interface body specifies all of the procedure's characteristics and these" by "The procedure characteristics specified by an interface body".

[206:21+] Add a new section 12.5.2.1 subsidiary to section 12.5.2 and renumber subsequent subsections:

12.5.2.1 Submodule procedures

A **submodule procedure** is a module procedure for which the interface is declared in a parent module (11.3) or submodule (11.3.1), and the body is defined in a submodule of that parent program unit. A submodule procedure body is logically a continuation of its procedure interface declaration in the parent module or submodule; it does not access the interface by host association.

A submodule procedure is accessible in its parent module or submodule. If the parent program unit is a module, and the procedure declared in the *submodule-specification-part* is public, it can be accessed by use association.

Note

It is possible to place specifications in a submodule declaration that do not contribute to specification of the interface. Unlike in an interface body, these specifications are part of the procedure.

[206:34] In section **12.5.2.2 Function subprogram** change the first line of the syntax rule (R1216) for *function-subprogram* to:

function-subprogram **is** *function-header*

[206:38+] In section **12.5.2.2 Function subprogram** add the following before the syntax rule (R1217) for *function-stmt*:

function-header **is** *function-stmt*
or *submodule-function-stmt*

[206:42+] In section **12.5.2.2 Function subprogram** add the following after the syntax rule (R1217) and constraint for *function-stmt*:

submodule-function-stmt **is** SUBMODULE FUNCTION *function-name* ■
■ [RESULT (*result-name*)]

Constraint: A *submodule-function-stmt* shall not appear except within a submodule.
Constraint: The *function-name* shall be declared, in a *submodule-procedure-declaration* in a *submodule-*

specification-part of the parent module or submodule of the submodule in which the *submodule-function-stmt* appears, to be a function that is not a dummy procedure. The *submodule-specification-stmt* that introduces that *submodule-specification-part* shall name the submodule in which the *submodule-function-stmt* appears.

Constraint: The *result-name* shall not be specified in both the *procedure-interface-declaration* and the *submodule-function-stmt*.

[207:14+] In section **12.5.2.2 Function subprogram**, in the last constraint after the syntax rule (R1220) for *end-function-stmt*, change *function-stmt* to *function-header*.

[208:25] In section **12.5.2.3 Subroutine subprogram** change the first line of the syntax rule (R1221) for *subroutine-subprogram* to:

subroutine-subprogram **is** *subroutine-header*

[208:29+] In section **12.5.2.3 Subroutine subprogram** add the following before the syntax rule (R1222) for *subroutine-stmt*:

subroutine-header **is** *subroutine-stmt*
or *submodule-subroutine-stmt*

[208:32+] In section **12.5.2.3 Subroutine subprogram** add the following after the syntax rule (R1222) and constraint for *subroutine-stmt*:

submodule-subroutine-stmt **is** SUBMODULE SUBROUTINE *subroutine-name*

Constraint: A *submodule-subroutine-stmt* shall not appear except within a submodule.

Constraint: The *subroutine-name* shall be declared, in a *submodule-procedure-declaration* in a *submodule-specification-part* of the parent module or submodule of the submodule in which the *submodule-subroutine-stmt* appears, to be a subroutine that is not a dummy procedure. The *submodule-specification-stmt* that introduces that *submodule-specification-part* shall name the submodule in which the *submodule-subroutine-stmt* appears.

[208:41+] In section **12.5.2.3 Subroutine subprogram**, in the last constraint after the syntax rule (R1224) for *end-subroutine-stmt*, change *subroutine-stmt* to *subroutine-header*.

[282:24] In **14.6.1.3 Host association** add "a procedure interface declaration that is not an interface body, a subsidiary submodule of a module" after "module subprogram" in the first sentence.

[283:8] In the last line of the first paragraph after the list of entities that can become inaccessible as a result of host association, replace "subprogram" by "scoping unit".

[298:30+] Add **parent module or submodule** to the glossary:

parent module or submodule (11.3): A *module* or *submodule* that has *submodules*.

[299:20] In the glossary entry for **program unit**, add "a *submodule*" after "*module*".

[300:24+] Add **submodule** and **submodule procedure** to the glossary:

submodule (11.3.1): A *program unit* that is logically an extension of a *module* or *submodule*, but cannot be accessed directly by *use association.*

submodule procedure (12.5.2.1): A *module procedure* for which the interface is declared in a *parent module or submodule*, and the body is defined in a *submodule* of that parent program unit.

[334:17+] Add a new section subsidiary to section **C.8.3 Examples of the use of modules**:

C.8.3.9 Modules with submodules

This example illustrates a module, `color_points`, with a submodule, `color_points_a`, that in turn has a submodule, `color_points_b`. Public entities declared within `color_points` can be accessed by use association. The module `color_points` does not have a *contains-part*, but a *contains-part* is not prohibited. The module `color_points` could be published as definitive specification of the interface, without revealing trade secrets contained within `color_points_a` or `color_points_b`.

```
module color_points
  type color_point
    private
    real :: x, y
    integer :: color
  end type color_point
submodule :: color_points_a ! Interfaces for procedures with separate
                            ! bodies in the submodule color_points_a
  subroutine color_point_del ( p ) ! Destroy a color_point object
    type(color_point) :: p
  end subroutine color_point_del
  real function color_point_dist ( a, b ) ! Distance between two color_point objects
    type(color_point) :: a, b
  end function color_point_dist
  subroutine color_point_draw ( p ) ! Draw a color_point object
    type(color_point) :: p
  end subroutine color_point_draw
  subroutine color_point_new ( p ) ! Create a color_point object
    type(color_point) :: p
  end subroutine color_point_new
end module color_points
```

The only entities within `color_points_a` that can be accessed by use association are procedures declared in submodule specification parts of `color_points` (in this case, there is only one submodule specification part). If the procedures' bodies are changed but their interfaces are not, the interface from program units that access them by use association is unchanged. If the module and submodule are in separate files, utilities that examine the date of modification of a file would notice that changes in the module could affect the

translations of program units that access the module by use association, but that changes in submodules could not.

The variable `instance_count` is not accessible by use association of `color_points`, but is accessible within `color_points_a`, and its submodules.

```
submodule(color_points) color_points_a ! Submodule of color_points
  integer, save :: instance_count = 0
! Procedure names are in alphabetical order
contains ! Invisible bodies for public interfaces declared in the module
  submodule subroutine color_point_del ! ( p )
    instance_count = instance_count - 1
    deallocate ( p )
  end subroutine color_point_del
  submodule function color_point_dist result(dist) ! ( a, b )
    dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
  end function color_point_dist
submodule :: color_points_b
  submodule subroutine color_point_draw ! ( p )
  ! "submodule" prefix indicates the interface is defined in the parent, not here.
  ! Being in a submodule specification part means the body is not here, either.
contains
  submodule subroutine color_point_new ! ( p )
    instance_count = instance_count + 1
    allocate(p)
  end subroutine color_point_new
submodule :: color_points_b ! continuation of above.
                            ! Interface for a procedure with a separate
                            ! body in submodule color_points_b
  subroutine inquire_palette ( pt, pal )
    use palette_stuff       ! palette_stuff, especially submodules
                            ! thereof, can access color_points by use
                            ! association without causing a circular
                            ! dependence because this use is not in the
                            ! module.  Furthermore, changes in the module
                            ! palette_stuff are not accessible by use
                            ! association of color_points
    type(color_point), intent(in) :: pt
    type(palette), intent(out) :: pal
  end subroutine inquire_palette
end submodule color_points_a
```

The subroutine `inquire_palette` is accessible within `color_points_a` because its interface is declared within a submodule specification part therein. It is not, however, accessible by use association, because its interface is not declared in a submodule specification part of the module, `color_points`. Since the interface is not

declared in the module, changes in the interface cannot affect the translation of program units that access the module by use association.

```
submodule(color_points_a) color_points_b ! Subsidiary**2 submodule
contains ! Invisible body for interface declared in the parent submodule
  submodule subroutine color_point_draw ! ( p )
  ! "submodule" prefix indicates the interface is defined in the parent, not here.
  ! Being in a contains part means the body is here.
    type(palette) :: MyPalette
    ...; call inquire_palette ( p, MyPalette ); ...
  end subroutine color_point_draw
  submodule subroutine inquire_palette
  ! "use palette_stuff" not needed because it's in the parent submodule
    ... implementation of inquire_palette
  end subroutine inquire_palette
  subroutine private_stuff ! not accessible from color_points_a
    ...
  end subroutine private_stuff
end submodule color_points_b

module palette_stuff
  type :: palette ; ... ; end type palette
contains
  subroutine test_palette ( p )
  ! Draw a color wheel using procedures from the color_points module
    type(palette), intent(in) :: p
    use color_points ! This does not cause a circular dependency because
                     ! the "use palette_stuff" that is logically within
                     ! color_points is in the color_points_a submodule.
    ...
  end subroutine test_palette
end module palette_stuff
```

There is a `use palette_stuff` in `color_points_a`, and a `use color_points` in `palette_stuff`. The `use palette_stuff` would cause a circular reference if it appeared in `color_points`. In this case it does not cause a circular dependence because it is in a submodule. Submodules are not accessible by use association, and therefore what would be a circular appearance of `use palette_stuff` is not accessed.

```
program main
  use color_points
  ! "instance_count" and "inquire_palette" are not accessible here
  ! because they are not declared in the "color_points" module.
  ! "color_points_a" and "color_points_b" cannot be accessed by
```

```
  ! use association.
  interface ( draw ) ! just to demonstrate it's possible
    module procedure color_point_draw
  end interface
  type(color_point) :: C_1, C_2
  real :: RC
  ...
  call color_point_new (c_1)        ! body in color_points_a, interface in color_points
  ...
  call draw (c_1)                   ! body in color_points_b, specific interface
                                    ! in color_points, generic interface here.
  ...
  rc = color_point_dist (c_1, c_2)  ! body in color_points_a, interface in color_points
  ...
  call color_point_del (c_1)        ! body in color_points_a, interface in color_points
  ...
end program main
```